Building Intrusion Tolerant Applications[*]

Thomas Wu
tjw@cs.stanford.edu

Michael Malkin
mikeym@stanford.edu

Dan Boneh[†]
dabo@cs.stanford.edu

Abstract

The ITTC project (Intrusion Tolerance via Threshold Cryptography) provides tools and an infrastructure for building intrusion tolerant applications. Rather than prevent intrusions or detect them after the fact, the ITTC system ensures that the compromise of a few system components does not compromise sensitive security information. To do so we protect cryptographic keys by distributing them across a few servers. The keys are never reconstructed at a single location. Our designs are intended to simplify the integration of ITTC into existing applications. We give examples of embedding ITTC into the Apache web server and into a Certification Authority (CA). Performance measurements on both the modified web server and the modified CA show that the architecture works and performs well.

1 Introduction

To combat intrusions into a networked system one often installs intrusion detection software to monitor system behavior. Whenever an "irregular" behavior is observed the software notifies an administrator. In this paper we study a complementary approach we call *intrusion tolerance*. Rather than prevent or detect intrusions after the fact, we provide tools that limit the amount of damage an intruder can cause. Our goal is to ensure that an attacker who penetrates a few system components cannot compromise total system security. To defeat our system an attacker must either penetrate multiple components in a short amount of time or frequently penetrate a certain component. Either way such large scale attacks are much easier to detect than single isolated attacks.

To describe our approach we consider a web server as an example. To enable secure connections to the

---

[*] Project URL: `http://d.n...b.`

see in Section 7 the performance penalty for doing so is small compared to the total time to establish an SSL connection. Consider again the example of a bank's system. By placing the share servers in a secure subnet and ensuring that they only accept connections from the bank's web servers one can strengthen the security of the bank's private key. Furthermore, our share servers can service multiple web servers. Hence, a bank maintaining dozens of web servers need not store a private key on each of them. Instead, the private keys can be stored (in shared form) on the share servers. This way a small number of break-ins will not compromise any of the private keys.

As an additional benefit, our system provides high availability of private keys. Even if a few of the share servers crash (by accident or as a result of an attack) and all data stored them is lost, the remaining active share servers automatically compensate for the corrupt ones. Furthermore, the system can recover by redistributing valid shares to the corrupted share servers.

The paper describes the design and implementation of our system including the prototype applications we built. We provide detailed performance measurements on both the modified web server and the modified CA to show that the architecture works and performs well.

2 System components

To protect private keys used by applications such as a web server or a CA, our system distributes shares of private keys among a number of share servers. In this section we describe the system components. We refer to applications using the share servers as *clients*. Web servers and CA's are example clients. Figure 1 illustrates the interaction between the various components.

Share server The share servers are implemented as daemons running on different machines. They hold shares of the private keys belonging to the clients they serve. These shares reveal no information about the client's private key. A share server can manage multiple keys and serve a large number of clients. We envision the total number of share servers being less than ten. Clearly it is desirable that an attacker not be able to compromise the share servers. However, the intrusion tolerant design of our system ensures that even if a few of the share servers are penetrated and the secrets stored on them are exposed or corrupted there is no compromise to overall system security. In other words, an attacker learns nothing from penetrating a few of the share servers. The system identifies corrupt share servers and can be instructed to take appropriate action to refresh the secrets stored on them.

Client. A client is any application that makes use of our client side Threshold LiBrary (the TLB). When a client connects to the share servers it first authenticates itself as an authorized client. It then interacts with the share servers to sign a message or decrypt a given ciphertext using the shared private key stored on the share servers. One concern is that an attacker can penetrate a client and expose the client's authentication key. The attacker can then masquerade as the client and fool the share server into applying a shared key. In Section 5.2 we describe a mechanism for detecting such an attack. At any rate, the attacker cannot expose the shared private key (as is the case when the private key is stored on the client).

Administrator. We provide a central administration utility to administer all the share servers. The administrator utility manages the various keys stored on the share servers. It can shutdown or suspend the servers if necessary, or instruct the servers to take appropriate action if some of the servers have been penetrated. The utility is provided for convenience. If for some reason central administration is not desired the share servers can be controlled locally.

Monitor. We built a monitor utility for testing and demonstrating our system. The monitor is a single daemon that collects information from all key servers and clients. Essentially, the share servers and clients send data to the monitor (via UDP) telling it what they are doing at any given moment. For instance, a client may tell the monitor that it is asking servers number 1, 2 and 4 to apply key number 23 to sign a message. A moment later share servers 1, 2 and 4 will each tell the monitor that they are applying shares of key number 23 to generate a signature, and so on. The monitor collects all this information and sends it to a Java applet that displays it in a human readable form. During actual deployment the monitor can be easily disabled by setting the appropriate switch in the system's configuration file. The administrator can still monitor system behavior by viewing each server's log file.

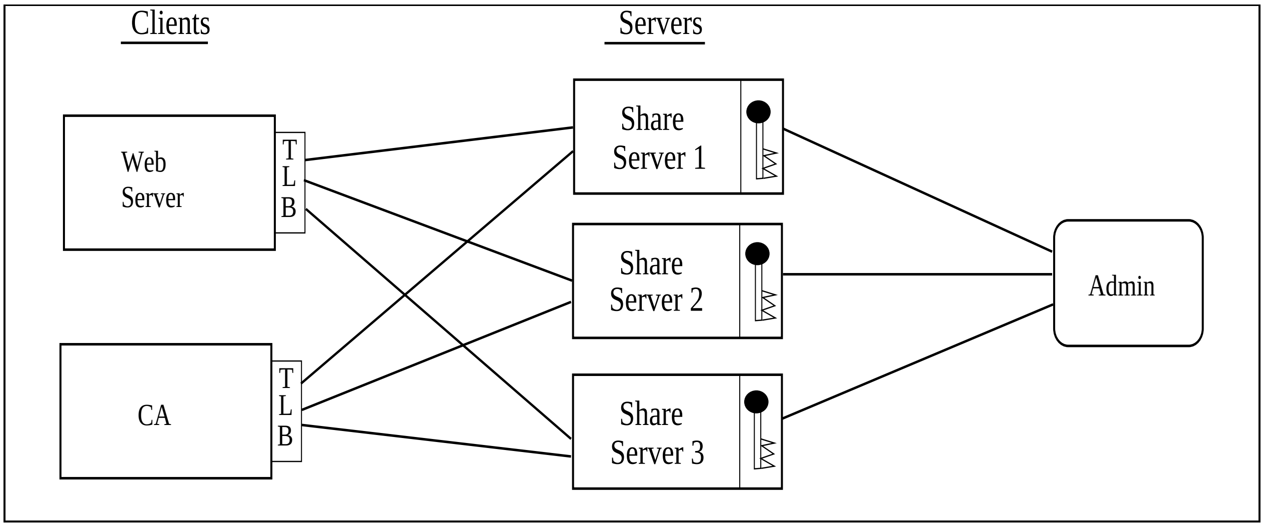

Figure 1: System components

2.1 Interaction between components

Each share server manages shares of the private key belonging to its clients. These shares are stored on disk encrypted using the administrator's passphrase. When the share server is started (possibly as part of the system's bootup script) it enters *suspended* mode. In suspended mode the share server refuses to accept any connections except a connection from the administrator. When all share servers are started, the admin program can be used to activate the servers by sending the passphrase to all of them at once. Connections between the admin program and share servers are always protected by SSL using mutual authentication. Using a hash of the administrator's passphrase, each server loads all shares available to it and enters *active* mode. The server is then ready to serve its clients.

In active mode a share server can accept connections from its clients and the administrator. The first step in any connection to a share server is mutual authentication and key exchange using SSL. The second step is an integrity check of the peer as described in Section 5.2. After a secure link is established, requests to the share server use the following protocol:

Request:	COMMAND $\langle opcode \rangle$ $\langle data \rangle$
Response:	RESPONSE $\langle status \rangle$ $\langle data \rangle$

The response status is either (1) OK indicating a successful completion, (2) SUSPENDED indicating that the share server is in suspended mode, or (3) a detailed error-code.

To understand the interaction between the system components we describe some of the opcodes supported by the share servers.

Connection from client. Each connection from a client causes the share servers to create a new thread (up to a specified maximum number of threads). These threads last as long as the client keeps open the connection to the share server, reducing the overhead of repeatedly opening and closing connections. A client can send a number of opcodes instructing the servers to apply a shared key. The interaction that takes place as a result of these commands is described in Section 4.

SIGN. Instructs the share server to apply its share of the private key for the purpose of generating a signature. The parameters include a digest of the message to be signed. We note that prior to taking part in the signing protocol described in Section 4 the key-servers verify that the message to be signed follows the PKCS1 format [11].

DECRYPT. Instructs the share server to apply its share of the private key for the purpose of decrypting a ciphertext.

HASH-SIGN. Instructs the share server to apply its share of the private key for the purpose of generating a hash of a signature. The purpose of this opcode is explained in the next section.

Connection from the admin. Some of the commands from the administrator include:

SHUTDOWN, SUSPEND, ACTIVATE. Manage share server modes. Shutdown causes all share server daemons to exit.

GENKEY. The admin can instruct the share servers to generate a new shared key. In our system private keys are *never* constructed in a single location. When an RSA key is generated it is generated in shared form. This uses a protocol due to Boneh and Franklin [3]. We use the implementation of shared RSA key generation described in [10]. The shares of the new private RSA key are stored on the share servers (encrypted with the admin password). The new public key is sent to the admin and is saved on the admin's machine. Authorized clients can then connect to the share servers to generate signatures or decrypt incoming messages using the new key.

A new public key is usually certified by a CA before it can be properly used. To obtain a public key certificate for the new key the admin must generate a *certificate request* to be sent to a CA. The format of a certificate request (specified in PKCS10) includes a *self signature* on the request. In other words, the newly generated *private* key must be used to sign the public key certificate request. The self signature ensures that the entity requesting the certificate has the corresponding private key. Unfortunately, in our case, the admin does not have the private key. In fact, no one has the private key — the private key is *always* stored in shared form on the share servers. Consequently, to generate the certificate request, the admin must connect to the share servers *as a client* and ask them to sign the request. Once the admin obtains the self signed request he can forward it to the CA. The CA will send back the certificate. We provide a utility that enables the admin to generate a self signed certificate request, as described in Section 6.1.

REFRESH. Suppose the administrator suspects the system is under attack and some share servers are compromised. In this case, the admin can instruct all share servers to *refresh* their shares of the private keys. Refreshing the shares does not change the private key. It simply generates a new independent *sharing* of the private key. Suppose an attacker obtains the shares of a private key stored on one of the servers. Once the admin refreshes the shares of the private key the information in the attacker's hands becomes useless.

Refreshing is also used when shares stored on a share server are corrupted or lost. Corruption or loss could happen due to a denial of service attack or a simple server crash. Refreshing the shares causes the uncorrupted servers to generate new valid shares for the corrupted server. Consequently, the system gracefully tolerates corruption or loss of a few shares.

3 Key management

We begin by explaining how one shares a private RSA key among a number of share server so that the key can be used without ever having to reconstruct it. We then describe the structure and PEM format used to store these shared keys.

3.1 Sharing RSA keys

Recall that an RSA private key consists of a modulus N and a secret exponent d. The modulus N is a product of two large primes, and d is a positive integer less than N. To decrypt a ciphertext C one computes $C^d \bmod N$. Similarly to sign a message digest M one computes $M^d \bmod N$. Hence, both operations require an exponentiation to the power d modulo N. Without the secret exponent d it is believed to be hard to either decrypt ciphertexts or generate signatures.

We show how a private RSA key can be broken up into a number of pieces (shares). Each share can be stored on a separate server and yet the private key can be used without having to reconstruct the secret. The basic idea, due to Frankel [6], is to pick random numbers d_1, d_2, d_3 in the range $[-N, N]$ so that $d_1 + d_2 + d_3 = d$. We then store share d_i on share server number i, for $i = 1, 2, 3$. Note that an attacker who breaks into any two of the three servers learns nothing about the private key d. All three servers must be compromised to obtain d.

When a client wishes to apply the key to sign a message M it sends M to all three servers. Each server applies its own share d_i to obtain $S_i = M^{d_i} \bmod N$ and sends the result S_i back to the client. The client obtains S_1, S_2, S_3 from the three servers. It multiplies them to obtain the signature

$$d = d_1 + d_2 + d_3$$
$$d = d_4 + d_5 + d_6$$

Server 1	Server 2	Server 3	Server 4
d_1	d_2	d_3	d_3
d_4	d_4	d_5	d_6

3-out-of-4 sharing

$$d = d_1 + d_2$$
$$d = d_3 + d_4$$

Server 1	Server 2	Server 3	Server 4
d_1	d_1	d_2	d_2
d_3	d_4	d_3	d_4

2-out-of-4 sharing

Figure 2: Additive sharings of a private RSA key d.

$S = S_1 \cdot S_2 \cdot S_3 \bmod N$. Since $d = d_1 + d_2 + d_3$ we have that $S = M^d \bmod N$ as required. Note that the private key d was never reconstructed in order to be used. In addition, there is no communication between the share servers. The only interaction is between the client and each of the servers.

Clearly this approach generalizes to distributing a private RSA key among k servers. Even if $k-1$ of the shares are exposed, an attacker learns nothing about the private key d. Since all k share servers must be involved in applying to key we call this sharing a k-out-of-k sharing.

3.1.1 t-out-of-k sharing

There are a number of problems with the approach described above. Most importantly, it is not fault-tolerant. If one of the share servers crashes the entire system goes offline. If one of the share servers loses its share, the private key is lost forever. For this reason, we use a t-out-of-k sharing of the secret key; any t of the share servers can be used to apply the key. For instance, if we use a 3-out-of-4 sharing no harm is done if one of the share servers is taken offline. In addition, a break-in into any two servers reveals no information about the private key.

Typically a t-out-of-k sharing is achieved using Shamir's classic secret sharing [14]. Unfortunately, when using Shamir's secret sharing the keys must be reconstructed before they can be used. This is inappropriate for our purposes. Instead, we use a combinatorial construction to obtain a t-out-of-k sharing of d from several t-out-of-t sharings of d. We give two examples in Figure 2. In both examples all the d_i's are random integers in the range $[-N, N]$ satisfying the stated equalities. Each server stores multiple d_i's as indicated by the column corresponding to that server. Observe that in the example on the left, any three servers can apply the key while any two servers learn nothing about d. The example on the right is more fault tolerant, but compromising two servers reveals the key. A compromise of a single server reveals nothing about the key. More generally, we implemented an algorithm that constructs tables as above for a t-out-of-k sharing for any t and k. The algorithm is based on ideas from [1, 4].

When a client requests that the servers apply a private key, it specifies which coalition of t servers is used. Based on the coalition, each server locally decides which of its d_i's it will use and sends the resulting S_i back to the client. The client multiplies the t responses and obtains the signature S. The client then uses the public key to check the validity of S as a signature. This is done to ensure that all share servers responded properly. In Section 4 we explain how we deal with corrupt share servers that apply their private shares incorrectly.

3.2 Key structures and key storage

Our system manages three types of keys: (1) a standard RSA public key, (2) a private share stored on each share server, and (3) a public shared key stored on each client. The public key stored on the clients contains the public RSA key plus some additional public information. We describe each of these keys below.

The SSLeay package [16] supports reading and writing both public and private keys in PEM format. Our private shares and public shared keys are represented internally as extensions of the standard RSA key data structure. On disk, we support a PEM-encoded ASN.1 format similar to that used for RSA keys.

Public key This is a standard RSA public key made up of an RSA modulus N and a public exponent e. It is managed by the standard RSA functions provided in SSLeay. Note that

Figure 3: Shared key file formats.

type:	INT	INT	INT	INT	INT	INT	INT	INT	$\cdots$	INT
data:	version	N	e	k	t	w	d_1	d_2	$\cdots$	d_w

(a) Private key file format.

type:	INT	INT	INT	INT	INT	INT	INT	INT	INT	$\cdots$	INT
data:	version	N	e	k	t	g	u	g^{d_1}	g^{d_2}	$\cdots$	g^{d_u}

(b) Public shared key file format.

an outsider communicating with the clients is unaware that the corresponding private key is stored in shared form.

Private share The private share stored on each share server contains the modulus N and a number of private d_i's. For a t-out-of-k shared key the private share file format (as ASN.1) is shows in Figure 3a. Each share server is given w shares $d_1, \ldots, d_w$. Note that the private share file does not contain the optional values $d \bmod p-1$, $d \bmod q-1$, or $q^{-1} \bmod p$, normally used to optimize RSA computations, where $N = pq$. After all, none of the parties can construct these values.

Public shared key The format of the public key file stored on each client is shown in Figure 3b (all arithmetic is done modulo N). Here u is the total number of g^{d_i}'s stored on the client. The values g and $g^{d_i} \bmod N$ are used to detect incorrect (or possibly compromised) private share operations by the share servers, as discussed in the next section. For each share d_i on each share server, there is a g^{d_i} in the public shared key. For example, in the 3-out-of-4 sharing of the previous section the public shared key contains six entries, so $u = 6$.

4 Using a shared key

We are now ready to describe the interaction that takes place when a client requests the share servers to apply a private key to a message M. Throughout the interaction the client keeps two bit-vectors `offline` and `corrupt`, indicating which of the share servers are currently offline and which have been found to be corrupt. We assume the client wishes to use a t-out-of-k shared key.

Init The client sets `offline` and `corrupt` to zero.

Step 1: The client picks a *random* coalition of t servers (out of k) that are neither offline nor corrupt.

Step 2: The client sends the following message to each of the servers in the coalition:

COMMAND	SIGN or DECRYPT	M	coalition	key-ID	corrupt	offline

where `key-ID` is the ID of the key the servers should apply. The `key-ID` is simply a 32 bit hash of N and e.

Step 3: Based on the coalition being used, each of the servers extract the appropriate d_i from its private key share of `key-ID`. It then locally computes $S_i = M^{d_i} \bmod N$ and send S_i back to the client. Note that if a signature is being requested, the client first verifies that the message M is in PKCS1 format, and rejects the request if not.

Step 4: The client collects all the S_i's and computes $S = \prod_{i=1}^{t} S_i \bmod N$. If some of the servers were found to be offline, the `offline` bit vector is updated and the process is restarted at Step 1.

Step 5: If $S^e = \pm M \bmod N$ the key was applied correctly and the process terminates.

Step 6: Otherwise, for each server in the coalition the client performs a zero-knowledge test to validate the server's response. The test is described in the next section. All servers that sent incorrect values are marked as corrupt and the process is restarted at Step 1.

If at Step 1 there are less than t servers that are neither corrupt nor offline the process fails. Note that the identity of corrupt share servers is sent to *all* share servers. This enables the servers to notify the administrator, who can take appropriate action to refresh the shares stored on the corrupt servers.

Load balancing When many different clients use the same share servers, the load on the servers may hurt overall system performance. Fortunately, our choice of a random coalition in Step 1 provides for *automatic load balancing* among the servers. When a 3-out-of-5 key is used each application of the key is likely to make use of a different set of three servers. Furthermore, by using a sufficiently low timeout period clients can avoid waiting for busy servers by switching to a different coalition. Due to the load balancing effect, a 3-out-of-5 sharing will result in a higher throughput than a 3-out-of-4 sharing.

4.1 Identification of corrupt servers

It remains to show how we identify corrupt servers in Step 6. This protocol is only used in the unlikely event one of the servers returns an incorrect response in Step 3. Recall that server i sent $S_i = M^{d_i} \bmod N$ back to the client where d_i is its share of the private key. By examining its own public shared key file the client obtains $V_i = g^{d_i} \bmod N$. To validate sever i's response the client does the following:

Step 6.1: The client picks random a and b in the range $[1, N]$ and computes $Z = M^a g^b \bmod N$.

Step 6.2: It sends the following to server i:

COMMAND	HASH-SIGN	Z		
	coalition	key-ID	corrupt	offline

Step 6.3: A legitimate server will respond with $A_i = H(Z^{d_i} \bmod N)$ where H is a cryptographic hash function. We use SHA-1.

Step 6.4: The client checks that $A_i = H[S_i^a V_i^b \bmod N]$. If not, the server is declared to be corrupt.

The above protocol is a simplification of a protocol from [7]. The following lemma shows that a share server who sends an invalid S_i will be caught with overwhelming probability. Furthermore, the protocol reveals no information about the server's secret shares. The lemma relies on the Small Order Assumption (SOA): there is no efficient algorithm that, given an RSA modulus $N = pq$, outputs an element x in $\mathbb{Z}_N$, where $x \neq \pm 1$, so that the order of x in $\mathbb{Z}_N$ is less than $N^{1/2}$.

Lemma 4.1 *Assuming SOA, a server who in step 2 sends any value other than* $\pm M^{d_i} \bmod N$ *will fool the above protocol with probability at most* $1/N^{1/2}$. *Furthermore, assuming the hash function used is a random function, the protocol is zero-knowledge (i.e. the client can simulate its interaction with the server).*

For a 1024 bit key, the probability that a server fools the client is less than $1/2^{512}$ — negligibly small. The hash function is used to prevent a malicious client from obtaining signatures on improperly formatted messages (i.e. not in PKCS1 format). For example, in a CA environment the DECRYPT command is disabled. Consequently, the servers can only be used to generate signatures and they only sign properly formatted PKCS1 messages. In this environment, the hash function prevents the client from using the validation test to obtain signatures on arbitrary messages.

We note that when $\gcd(p-1, q-1) = 2$ the SOA assumption is equivalent to assuming the hardness of factoring. However, since our servers cannot test this condition (no single server knows p and q) we use the SOA as the intractability assumption in Lemma 4.1.

5 Implementation details

5.1 Certificate formats

The ITTC system uses the Secure Socket Layer (SSL) to authenticate and encrypt all communication. The administrator issues a certificate to each entity (client, share server, and administrative utility) in the system, which must be in a recognized format. The CN (Common Name) field in each certificate contains a string that allows other parties to identify it as described in the following table:

Entity	ID String
Client	`[CLIENT` n`]`
Share Server	`[SERVER` n`]`
Administrator	`[ADMIN` n`]`

When establishing an SSL-secured connection, both peers must send their certificates to establish mutual authentication. Each peer verifies the certificate it receives and then parses it to extract the identifying string to ensure that the peer is authentic. If a certificate is not sent, if verification fails, or if the identity string does not match what was sent in the protocol, the connection fails.

5.2 Sequence numbers

Although client and server certificates thwart direct network snooping and impersonation attacks against the servers, the ITTC threat model requires the system to tolerate and resist successful attacks against individual clients and servers as well. If an attacker, for example, obtains the private key and certificate of a legitimate client, he could use it to obtain access to any shared keys that the client had access to. To limit the amount of damage under these circumstances, ITTC uses sequence numbers in the connection protocols to detect such compromises after the fact. Each client and server keeps a count of how many times it has been accessed by each other entity in the system, and all connections involve an exchange of sequence numbers to verify synchronization. An attacker who uses a stolen key and certificate will cause the sequence number at each server to be incremented without a corresponding change in the sequence number at the "victim" client's system. The next time that client attempts to access a server, the mismatch will be detected.

The actual multi-step sequence number exchange proceeds as follows. Assume that both client and server are currently synchronized at sequence number n initially:

1. The server sends n to the client and sets its own sequence number to $n + 1$.
2. The client expects n. It sends $n+1$ back to the server. It sets its own sequence number to $n+1$. (In this step, the client also accepts $n + 1$ from the server. If this happens, it sends $n + 2$ back to the server and sets its own sequence number to $n + 2$. The rest of the protocol proceeds with the client adding one to all its sequence numbers.)
3. The server expects $n + 1$. It sends $n + 2$ back to the client. It sets its own sequence number to $n + 2$.
4. The client expects $n + 2$. It sets its own sequence number to $n + 2$.

When the protocol completes successfully, both client and server hold sequence number $n + 2$. The advantage this protocol has over a simpler single-increment protocol is that it distinguishes between ordinary network failure and a genuine security breach. In a single-increment protocol, it is possible for a network failure to cause one party to increment its sequence number while the other does not. The off-by-one discrepancy in the sequence numbers would be interpreted as a security breach during subsequent interactions between these two parties. This double-increment protocol, on the other hand, increments sequence numbers by two for each successful interaction, while ensuring that the server's sequence number never drifts by more than one relative to the client's number. A network failure would result in at most an off-by-one discrepancy, which would then be resolved in step two of the protocol. A real security breach would cause the numbers to differ by two or more, presumably triggering a legitimate alarm.

5.3 Interface to TLB

To access the underlying ITTC functionality, client programs link against the TLB code and call functions in its external interface. Since it is possible to perform all the standard RSA cryptographic operations (encrypt, decrypt, sign, verify) with an ITTC-style RSA key, existing applications can be rewritten to use these keys by replacing conventional RSA calls with calls to TLB functions.

This approach has some drawbacks, most notably that it requires effort to go through each application to find RSA cryptographic calls and replace them with their ITTC counterparts. Fortunately, the design of SSLeay allows us to implement the TLB interface more elegantly. In SSLeay, RSA keys (type `RSA *`) contain not only the data corresponding to the keys (N, e, d, etc.) but also pointers to functions that perform the four basic RSA operations. This form of polymorphism allows us to reduce the entire TLB interface to a single function call:

```
RSA *ITTC_load_RSA_key(char *ittckeyfile);
```

This function returns an RSA object whose "function table" points to functions that perform ITTC-style decryption and signing (encryption and signature verification are handled just like an ordinary RSA key). Modifying an application to handle ITTC-style keys is now a matter of changing only one piece of code, namely the code that loads RSA keys. Once the key is loaded in this manner, application code cannot distinguish an RSA object obtained through ITTC from a conventional RSA key, and no other code needs to be modified.

Section 6 contains some examples of applications that used this technique to support ITTC-style shared keys. This object-oriented method of encapsulating functionality with data can be carried over to other cryptographic architectures (e.g. Java's JCA or Microsoft CAPI) that support provider interfaces.

6 Example applications

6.1 Intrusion tolerant certification authority

The ITTC-based certificate authority application signs incoming certificate requests with the appropriate shared private key. The application itself is based on Eric Young's `ca` certificate authority program, which is included with SSLeay. As described in Section 5.3, adapting the existing code to handle ITTC keys involved only a minimal change to the code, as seen in Table 1.

```
#ifndef ITTC
  BIO_read_filename(in,keyfile);
  pkey=PEM_read_bio_PrivateKey(in,NULL,NULL);
#else
  pkey=EVP_PKEY_new();
  rsa=ITTC_load_RSA_key(keyfile);
  EVP_PKEY_assign_RSA(pkey,rsa);
#endif
```

Table 1: ITTC Certificate Authority Code Patch

The modified certificate authority (`ittc_ca`), accepts the same inputs and command line arguments as the original program. For example, the command

```
ittc_ca -config ca.cnf -in request.req
    -out newcert.pem -keyfile sharedkey.rsk
```

verifies the self-signed standard-form certificate request in `request.req`, signs it with the shared RSA key `sharedkey.rsk`, and writes the PEM-encoded X509 result in `newcert.pem`. The file `ca.cnf` is an internal SSLeay config file.

In most applications where an ITTC private key is used, it is necessary to obtain a signed certificate for the corresponding public key from a well-known issuer (e.g. when using ITTC to secure a Web server's private key, see Section 6.2). The `ittc_req` program generates correctly-formatted, self-signed certificate requests from ITTC-style shared keys. The command

```
ittc_req -config ca.cnf -new
      -key sharedkey.rsk -out request.req
```

compiles a standard-form certificate request for the shared key `sharedkey.rsk` and applies the shared key once to sign it. The output (`request.req`) is indistinguishable from a certificate request generated from a conventional RSA key and can be sent off to any certificate authority (e.g. Verisign) to be signed.

6.2 Distributed web server

In addition to securing the private key for a certificate authority, ITTC can protect private keys for a Web server. The technique of Section 5.3 was applied to the source code for ApacheSSL 1.2.6 to produce an ITTC-enabled secure server. Since Apache already used SSLeay to handle its public-key cryptography, the entire extent of our modifications to Apache consisted of the three lines of C code shown in Table 2.

```
#ifdef ITTC
   pConfig->prsaKey =
           ITTC_load_RSA_key(szPath);
#else
   pConfig->prsaKey=RSA_new();
   PEM_read_RSAPrivateKey(f,
               &pConfig->prsaKey, NULL);
#endif
```

Table 2: ApacheSSL Code Modification for ITTC

With this change in place, Apache accepts ITTC-style private keys and treats them as if they were conventional RSA keys. An administrator simply generates an ITTC shared key, generates a self-signed certificate request with the `ittc_req` application (see Section 6.1), and sends this request to a recognized certificate authority. Once the signed certificate is received, Apache is configured to use the certificate and corresponding key with the following entries in its configuration file, `httpd.conf`:

```
SSLCertificateFile
     /usr/local/httpd/SSLconf/cert.pem
SSLCertificateKeyFile
     /usr/local/httpd/SSLconf/privatekey.rsk
```

Once the Web server is started up, it will present the certificate `cert.pem` to Web browsers and use the ITTC shared key `privatekey.rsk` to decrypt and/or sign responses.

7 Performance

The following timing measurements were taken on six Intel-based PC's. Two were running Solaris 2.6 at 333Mhz and four were running Windows NT at 450Mhz. The machines were connected via a 10base-T Ethernet. Clearly, a faster local network would improve our performance numbers.

Table 3 shows the latency and throughput for several key-sizes and sharings. Latency is the time that a client has to wait for a request to be serviced. Throughput is the number of requests that can be serviced per second. The table also lists the number of concurrent threads running on the *client*. The more threads run on the client the higher the load on the servers. Load balancing causes each share server to see some fraction of the requests. For example, when 10 threads are used the client makes 10 concurrent requests. If a 3-out-of-5 sharing is in use, each share server will service an average of $\frac{3}{5}$ of the total requests, or an average of 6 concurrent requests.

If the requests are processed serially, throughput depends only on latency, but when multiple clients connect to the servers at once load-balancing increases throughput. Increasing the number of simultaneous connections slows down each individual request but increases the throughput. Unless stated otherwise, the number of simultaneous connections was set to two.

A larger key corresponds to more communication and longer computations, so as the number of bits in the key increases, the latency increases and the throughput decreases. In every configuration, the 1024-bit key was significantly faster than the 2048-bit key.

Latency is affected by both the number of servers involved in a request and the total number of servers. For a fixed threshold, a larger total number of servers makes load-balancing more effective by distributing concurrent requests over more servers. This keeps the servers from being inundated by requests. When the total number of server is fixed, a higher threshold is likely to increase latency. To see this recall that a request completes only once all the servers in the coalition respond. With a larger coalition it is more likely that a busy server is included in the coalition. Hence, the time until the request completes is likely to increase.

The ratio of servers used to the total number of servers is the dominating factor in throughput. For example, with the 3-out-of-5 sharing, this ratio would be 0.6. As this ratio shrinks, a smaller portion of the servers is required per request, and more simultaneous requests can be processed with the same performance.

Table 3 also compares shared keys versus standard non-shared RSA keys. As expected, latency is larger when using a shared key than when using a non-shared key. The use of the shared key requires communication with share servers which is not required when using a non-shared key. Likewise, throughput is lower when using a shared key, but throughput doesn't degrade as much as latency because of load-balancing and multi-threading of the share servers.

As the number of simultaneous requests (threads) grows, latency also grows, because of the higher load on the servers. However, throughput improves because of load balancing and because the servers can service some threads while other threads are idle.

Table 4 shows what happens when some servers are taken offline. This is implemented by putting the share-servers in "suspended" mode. In this mode, a server can accept connections, but immediately closes them. Performance is identical when a server is simply shut down.

The TLB detects all offline servers in the coalition in one pass. When too many servers are offline, TLB

	thr.	no sharing		2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
1024	2	0.067s	29.2/s	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1024	10	0.070s	28.9/s	1.343s	7.14/s	1.180s	7.90/s	1.790s	4.83/s	1.755s	4.90/s
2048	2	0.370s	5.38/s	1.434s	1.38/s	1.268s	1.55/s	1.978s	1.01/s	1.975s	1.00/s

Table 3: Latency and throughput as a function of key size

	2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
0–offline	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1–offline	0.604s	3.26/s	0.532s	3.72/s	0.979s	2.01/s	0.887s	2.19/s
2–offline			0.763s	2.60/s			0.967s	2.03/s

Table 4: The effect of offline servers (1024-bit key)

returns an error code. For example, the 2-out-of-3 sharing will return an error code if two servers are offline, since the remaining server cannot service a request by itself.

Taking servers offline increases latency because some requests have to be issued multiple times, with different coalitions. Also, as servers are taken offline, load-balancing forces the remaining servers to service more requests, further increasing latency. Throughput also suffers when servers are taken offline, for the same reasons as latency.

Performance drops more when going from 0–offline to 1–offline than it does when going from 1–offline to 2–offline. 2–offline is worse than 1–offline since in the worst case requests may need to be issued three times until a valid coalition is found. As servers are taken offline, the sharings that use a larger fraction of the total servers degrade the most. For example, taking one server offline affects the 2-out-of-3 and 3-out-of-4 sharings more than the 2-out-of-4 sharing. This is because load balancing is more effective when the number of servers per request is small compared to the total number of servers.

Table 5 shows the effects of corrupt servers. Corrupt servers act just like normal servers, but they return invalid responses. In the tests used to find this data, corrupt servers compute the correct response, then return twice that value. When an invalid response is detected all servers in the coalition are checked for corruption. Hence, all corrupt servers in the coalition are detected in one pass.

Latency grows dramatically when corrupt servers are detected. There are two causes for this growth. First, just as with offline servers, requests may need to be issued multiple times. Second, when a corruption is found, each server in the corrupt coalition must be checked to see if it is corrupt (see Section 4.1). Comparing this chart to the previous chart, it becomes apparent that a corrupt server adds approximately 2.7 times latency penalty as an offline server in the same configuration. As with offline servers, throughput suffers as servers are corrupted, for the same reasons as latency.

Table 6 shows the performance of a web-server and certification authority, both using shared and non-shared private keys of size 1024-bits and 2048-bits. The web-server was tested by repeatedly establishing an SSL connection and issuing a `HEAD` request to a URL on the web-server. The number of threads listed in the table reflects the number of concurrent threads used by the test program when establishing connections to the web server.

In the web-server, both latency and throughput are worse with a shared-key than with a non-shared key, but not by much. With 10 simultaneous connections, latency is only 24% higher for a shared key and throughput is only 17% worse. For most applications, this slowdown is insignificant considering that SSL session establishment is only a small part of the interaction with the web server.

8 Related work

Fray, Deswarte and Powel [8] and Deswarte, Blain and Fabre [5] describe an encrypted file system where file keys are distributed using Shamir secret sharing across several key servers. Keys are reconstructed every time a file is accessed. In contrast, by using threshold RSA, the ITTC system never reconstructs long term private keys in a single location.

	2-out-of-3		2-out-of-4		3-out-of-4		3-out-of-5	
0–corrupt	0.411s	4.86/s	0.403s	4.95/s	0.639s	3.08/s	0.638s	3.13/s
1–corrupt	0.955s	2.06/s	0.823s	2.40/s	1.478s	1.31/s	1.334s	1.50/s
2–corrupt			1.536s	1.28/s			2.084s	0.95/s

Table 5: The effect of corrupt servers (1024-bit key)

	key-size	thr.	no sharing		2-out-of-3		2-out-of-4		2-out-of-4 1–offline	
web-server	1024	2	0.209s	9.50/s	0.644s	3.10/s	0.515s	3.84/s	0.535s	3.73/s
web-server	1024	10	1.097s	8.25/s	1.362s	6.85/s	1.503s	6.03/s	1.543s	5.66/s
web-server	2048	2	0.389s	5.05/s	1.527s	1.31/s	1.484s	1.34/s	1.509s	1.32/s
CA	1024	2	0.067s	29.2/s	0.411s	4.86/s	0.403s	4.95/s	0.532s	3.72/s
CA	2048	2	0.370s	5.38/s	1.434s	1.38/s	1.268s	1.55/s	1.749s	1.12/s

Table 6: Usage of ITTC in a CA and web server

The Ω system [12], built at AT&T, also uses threshold cryptography to protect private keys. Ω supported a Certification Authority (CA) used at AT&T. It was the first system to demonstrate the practicality of threshold cryptography. We note that Ω does not support distributed key generation, detection of corrupt servers or the ability to refresh shares in case a share server is compromised.

The proactive security toolkit [2] built at IBM focuses on using proactive security applied to DSS to protect private DSS signing keys. The system shares a DSS key among a number of servers, and proactively refreshes these shares once every predetermined time period (e.g. once a day). Interestingly, DSS and RSA have different sharing properties. RSA keys are easy to share (so that a signature can be generated without reconstructing the key), but are hard to generate distributively (so that none of the participants know the private key). On the other hand, DSS keys are easy to generate distributively, but are harder to use for threshold signatures.

Our particular implementation of threshold RSA is based on one of several possible algorithms. We chose an algorithm for threshold RSA that is best suited to handle a small number of share servers (i.e. less than six). For a larger number of servers one could use a recent algorithm due to Shoup [15].

9 Conclusions

The ITTC system enables applications to store their private keys in an intrusion tolerant fashion. Penetrating a few share servers reveals no information about the private key. Penetrating a client is detected through our use of sequence numbers and does not expose any shared keys. Our system eliminates single points of failure by never reconstructing a shared private key in a single location. Even key generation is done in shared form. The system detects and corrects offline and corrupt servers. For instance, one of the share servers can be taken offline for maintenance without affecting system behavior.

ITTC is easy to embed into existing applications. We built a web server and a CA in which private keys are managed using ITTC. Our performance figures show that the cost of using ITTC is reasonable. This is especially true in the case of a web server where session key exchange is only a small fraction of the total work performed by the server. Session key exchange is not done too frequently since both the browser and server cache session keys.

Acknowledgments

Research supported by DARPA/ITO. We thank Bill Aiello and Sivaramakrishnan Rajagopalan (Raj) for their help throughout the project.

References

[1] N. Alon, Z. Galil, M. Yung, “Dynamic re-sharing verifiable secret sharing against a mobile adversary”, in Proceedings of the 1995 European Symposium on Algorithms (ESA), pp. 523–537.

[2] B. Barak, A. Herzberg, D. Naor, E. Shai, “The proactive security toolkit and applications”, to appear in the 6th ACM Conference on Computer and Communications Security, 1999.

[3] D. Boneh, M. Franklin, “Efficient generation of shared RSA keys”, in Proceedings Crypto’ 97, pp. 425–439.

[4] Y. Desmedt, G. Di Crescenzo, M. Burmester, “Multiplicative non-abelian sharing schemes and their application to threshold cryptography”, Proceedings ASIACRYPT ’94, pp. 21–32.

[5] Y. Deswarte, L. Blain, J Fabre, “Intrusion tolerance in distributed computing systems”, Proceedings IEEE Symposium on Security and Privacy, Oakland, 1991, pp. 110–121.

[6] Y. Frankel, “A practical protocol for large group oriented networks”, Eurocrypt 89, pp. 56–61.

[7] Y. Frankel, P. Gemmel, P. MacKenzie, M. Yung, “Optimal-resilience proactive public-key cryptosystems”, Proceedings FOCS ’97, pp. 384–393.

[8] J. Fray, Y. Deswarte, D. Powell, “Intrusion tolerance using fine-grain fragmentation-scattering”, Proceedings IEEE Symposium on Security and Privacy, Oakland, 1986, pp. 194–201.

[9] P. Gemmel, “An introduction to threshold cryptography”, in CryptoBytes, a technical newsletter of RSA Laboratories, Vol. 2, No. 7, 1997.

[10] M. Malkin, T. Wu, D. Boneh, “Experimenting with shared RSA key generation”, Proceedings of the Internet Society’s 1999 Symposium on Network and Distributed System Security (SNDSS), pp. 43–56

[11] Public Key Cryptography Standards (PKCS), RSA Labs, available at `http://www.rsa.com/rsalabs/pubs/PKCS/`

[12] M. Reiter, M. Franklin, J. Lacy, R. Wright, “The Ω key management service”, Proceedings of the 3rd ACM conference on Computer and Communication Security, 1996.

[13] T. Rabin, “A simplified approach to threshold and proactive RSA”, Proceedings of Crypto’ 98.

[14] A. Shamir, “How to share a secret”, Comm. of the ACM, Vol. 22, 1979, pp. 612–613.

[15] V. Shoup, “Practical threshold signatures”, to appear.

[16] E. Young, SSLeay, `http://www.ssleay.org/`